IBM DEVOPS VIT

FINAL PROJECT

SeatSnap - A Ticketmaster-Style Site with DevOps Implementation

Submitted By:

-	
Name	AJITESH SHARMA
Registration No.	23BCI0027
Email	ajitesh.sharma2023@vitstudent.ac.in
Institution	Vellore Institute of Technology, Vellore
Degree	B.Tech CSE with specialization in Information Security
Batch	2023-2027

GitHub URL: https://github.com/AJ1312/SeatSnap-devops

1. Objective

The primary objective of this project is to design, develop, and deploy **SeatSnap**, a modern, intuitive web application that mirrors the functionality of a ticket marketplace like Ticketmaster. The core focus is not just on the application itself, but on demonstrating a robust, automated deployment strategy for the frontend. This is achieved by implementing a comprehensive **Continuous Integration and Continuous Deployment (CI/CD)** pipeline and leveraging advanced DevOps practices to ensure efficient, reliable, and secure software delivery.

2. Relevant Architecture & Technology Stack

The project's infrastructure is built on modern cloud-native and DevOps technologies to ensure scalability, resilience, and automation.

- **Frontend:** A modern web framework (Used: HTML/CSS/JS).
- **Containerization: Docker** is used to package the application and its dependencies into a standardized, portable container.
- **CI/CD Automation: Jenkins** orchestrates the entire build, test, and deployment pipeline.
- **Container Orchestration: Kubernetes (K8s)** manages the containerized application, handling scaling, self-healing, and service discovery.
- **Cloud Provider: Amazon Web Services (AWS)** ☁ provides the underlying cloud infrastructure and ***Built using EC2 Ubuntu instance***.
  - **Compute:** AWS EC2 Instances host the Kubernetes cluster nodes.
  - **Load Balancing:** An AWS Application Load Balancer distributes traffic evenly across application pods.
- **Monitoring & Observability:**

    - **Prometheus** scrapes and stores real-time metrics from the application and Kubernetes cluster.
    - **Grafana** visualizes the collected metrics through interactive dashboards for monitoring and alerting.

3. Features

Application Features

- **Event Discovery:** Users can browse a wide catalog of events, including concerts, sports games, and theater shows.
- **Detailed Event Information:** Users can view comprehensive details for each event, such as dates, times, venue information, and seating availability.

DevOps & Pipeline Features

- **Fully Automated CI/CD Pipeline:** A complete, end-to-end pipeline orchestrated by Jenkins automates the entire software delivery lifecycle.
- **Source Code Integration:** The pipeline is triggered automatically upon code commits to the **GitHub** repository.
- **Static Code Analysis: SonarQube** is integrated to perform automated code quality checks and maintain high standards.
- **Comprehensive Security Scanning:**
    - **Trivy** scans container images for known vulnerabilities before deployment.
- **Automated Testing:** The pipeline executes a suite of automated tests against the containerized application to validate functionality.
- **Automated Developer/Stakeholder Notifications:** Email notifications are automatically dispatched to keep relevant stakeholders informed of the pipeline's status (success or failure).

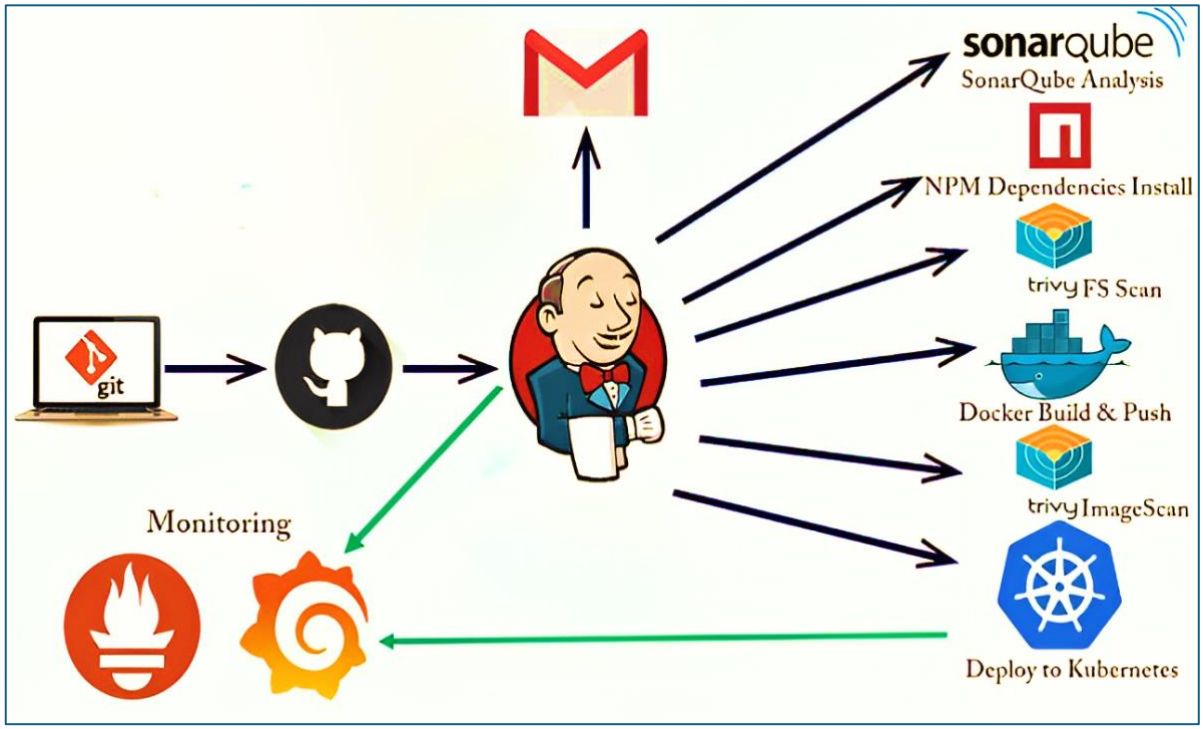
4. Future Scope & Roadmap

This project lays the foundation for a full-featured application. Future enhancements include:

- **Full Backend Integration:** Develop and integrate backend services for:
    - User Authentication and Profile Management
    - Secure Payment Processing
    - Real-time Ticket Inventory Management
- **Enhanced UI/UX:** Implement advanced search capabilities, event filtering (by date, genre, location), and interactive seat maps.
- **Push Notifications:** Add a notification system to alert users about upcoming events, price drops, or last-minute availability.
- **Third-Party API Integration:** Integrate with external APIs for venue maps, artist information, and event reviews.

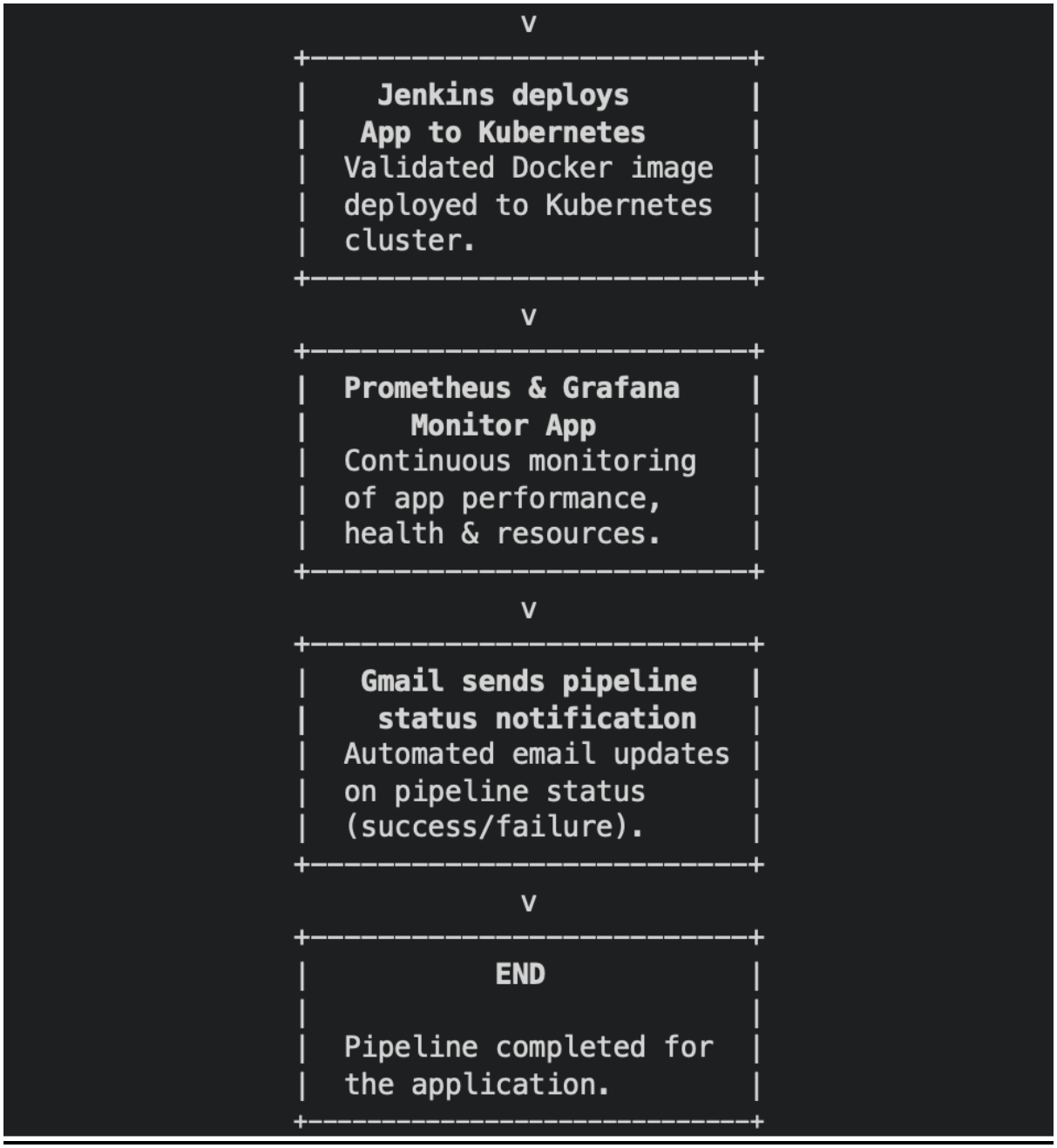
Explore Serverless Architecture: Leverage serverless functions (e.g., AWS Lambda) for specific, event-driven backend functionalities to optimize cost and scalability.

5. Application CI/CD Flow




```
+--------------------------+
|          START           |
|  Beginning of the CI/CD  |
|      process.            |
+--------------------------+
             v
+--------------------------+
|  Developer writes code   |
|      (using Git)         |
|  Code creation/modifica- |
|  tion on local machine;  |
|  Git for version control.|
+--------------------------+
             v
+--------------------------+
|  Push code to GitHub     |
|      Repository          |
|  Commit local changes &  |
|  push to centralized     |
|  GitHub repo.            |
+--------------------------+
             V
```


GitHub triggers Jenkins
(via webhook)
Webhook sends notification to Jenkins, starting the pipeline.
Jenkins initiates SonarQube Analysis
Static code analysis for bugs, vulnerabilities & code quality.
Jenkins installs NPM Dependencies
Installs all required project dependencies.
Jenkins performs Trivy FS Scan
Vulnerability scan of application codebase & dependencies.
Jenkins Builds & Pushes Docker Image
Creates container image of app & dependencies; pushes to registry.
Jenkins performs Trivy Image Scan
Security scan of Docker image for vulnerabilities.


v
Jenkins deploys App to Kubernetes
Validated Docker image deployed to Kubernetes cluster.
v
Prometheus & Grafana Monitor App
Continuous monitoring of app performance, health & resources.
v
Gmail sends pipeline status notification
Automated email updates on pipeline status (success/failure).
v
END
Pipeline completed for the application.

SnapSeat App Deployment and Step by Step Workflow

This document outlines the end-to-end process for deploying the SnapSeat application, from server setup and CI/CD automation to Kubernetes deployment and monitoring.

The following is divided into two parts among which

Part 1: Starting with workflow of CI/CD and getting our site running on AWS Instance.
Part 2: Setting up Grafana and Prometheus for Monitoring our instance.

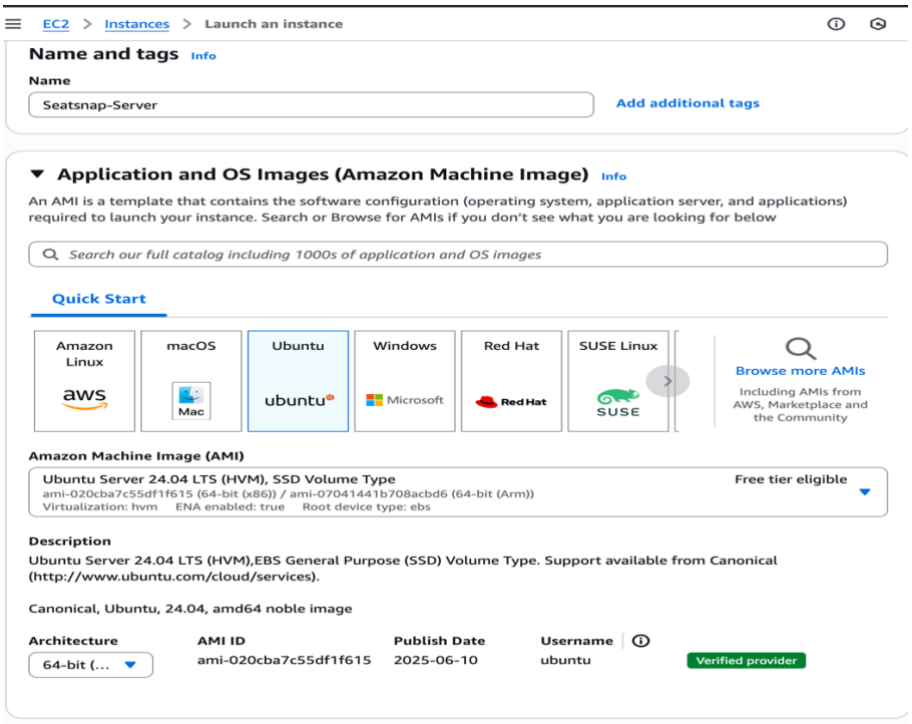
Part I: CI/CD Pipeline automation

This phase focuses on setting up the server, installing the necessary tools, and creating a CI/CD pipeline to build, test, and deploy the application as a Docker container.

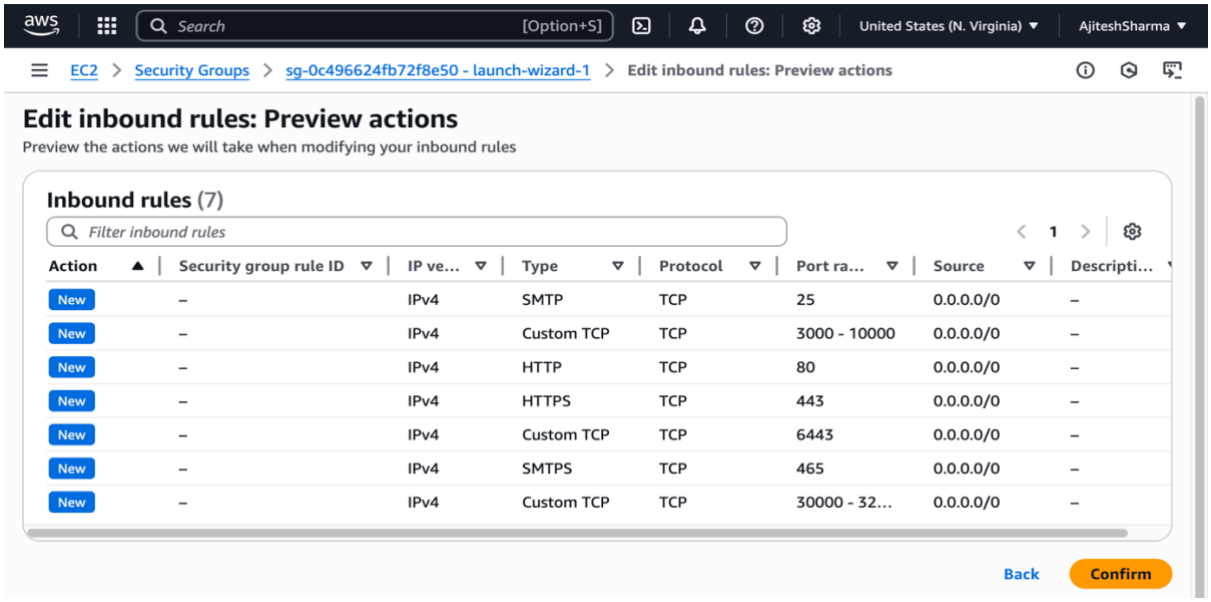
Step 1: Basic Setup & Server Preparation

The process begins by provisioning and configuring the primary server in AWS.

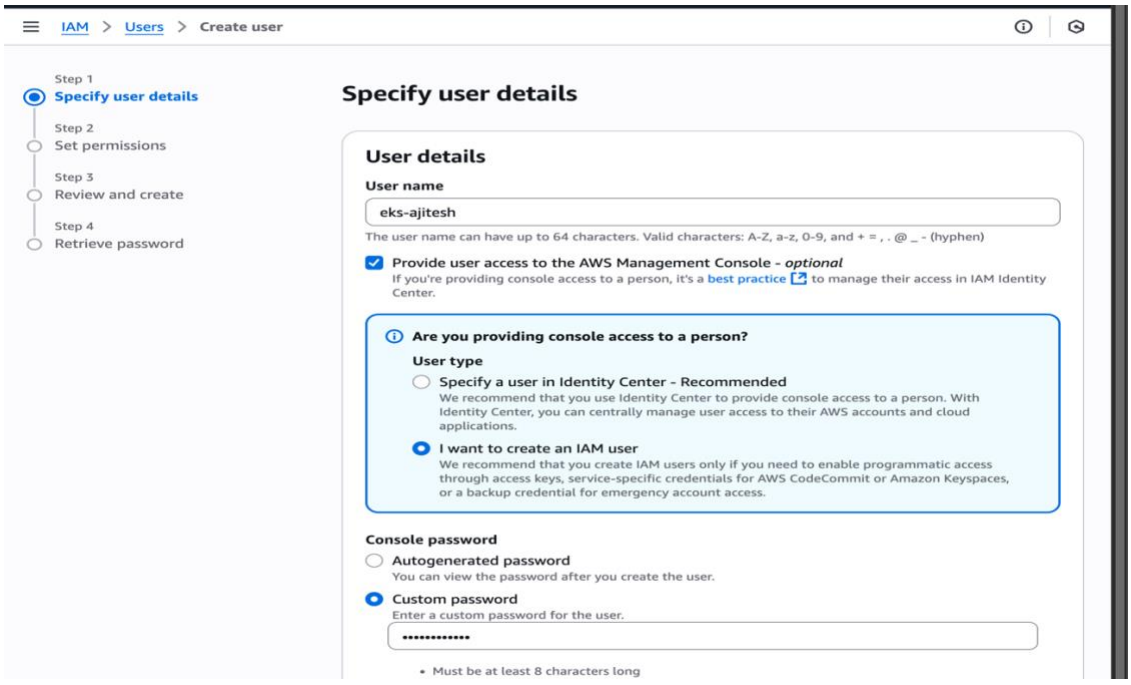
1. **Launch AWS EC2 Instance (SnapSeat-Server)**
    - A virtual server in AWS was created with the specifications: **Ubuntu 24.04, t2.large, 28GB Storage**. It was named SnapSeat-Server to clearly identify its purpose.



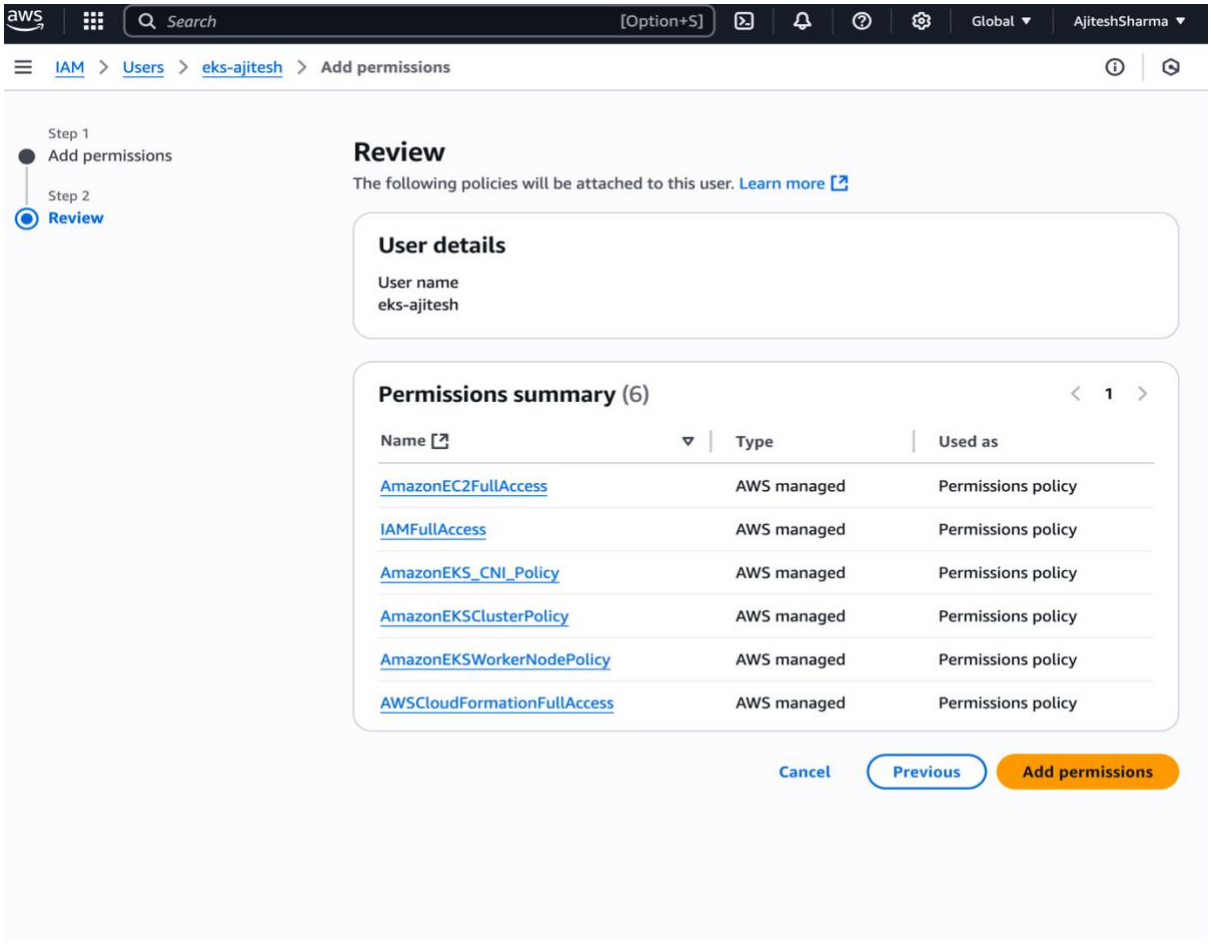
- **Configure Security Group:** The necessary ports were opened to allow traffic for the tools. This acts as a virtual firewall for the server, controlling inbound and outbound connections.



2. **Create an IAM User for EKS**
   - In the AWS IAM service, a new user named ajitesh-eks-user was created. This is a security best practice to avoid using the main AWS account for automated tasks, limiting potential security risks.
   - The required policies (e.g., AmazonEC2FullAccess, AmazonEKSClusterPolicy, etc.) were attached to grant this user the permissions needed to create and manage a Kubernetes cluster.
   - **Access Keys** for this user were generated and securely saved. These credentials are vital for configuring the AWS Command Line Interface (CLI) later.



Required Policies:



Step 2: Tools Installation on SnapSeat-Server

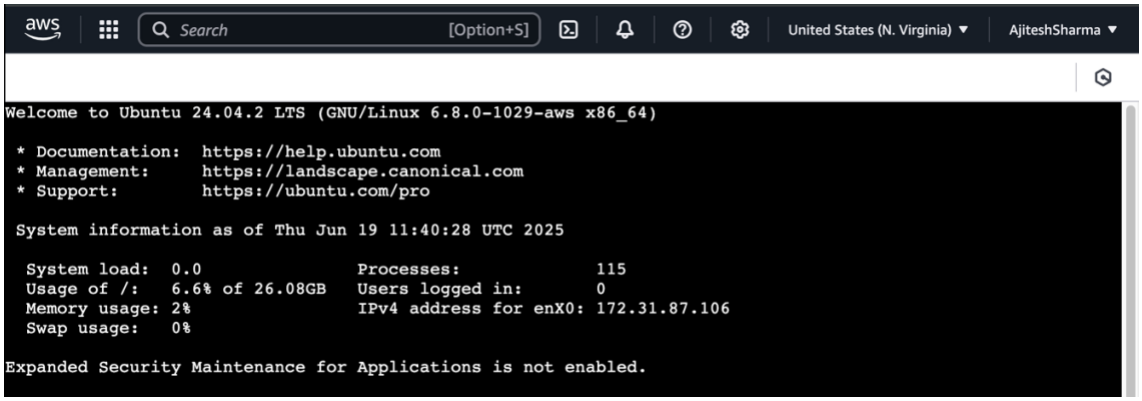
Next, a connection was established to the SnapSeat-Server, and the core DevOps tools were installed using shell scripts for automation.

1. **Opening up the Terminal Instance and Installing Tools**
    - Opening the terminal using AWS to connect to the newly created server.
    - sudo apt update: This command was run first to refresh the server's package lists, ensuring that the latest versions of software packages are available for installation.
    - The following scripts were created and run to install the toolset.

      1.vi <script_name>.sh was used to paste content, saved, and then

      2.sudo chmod +x <script_name>.sh was run to make the script executable

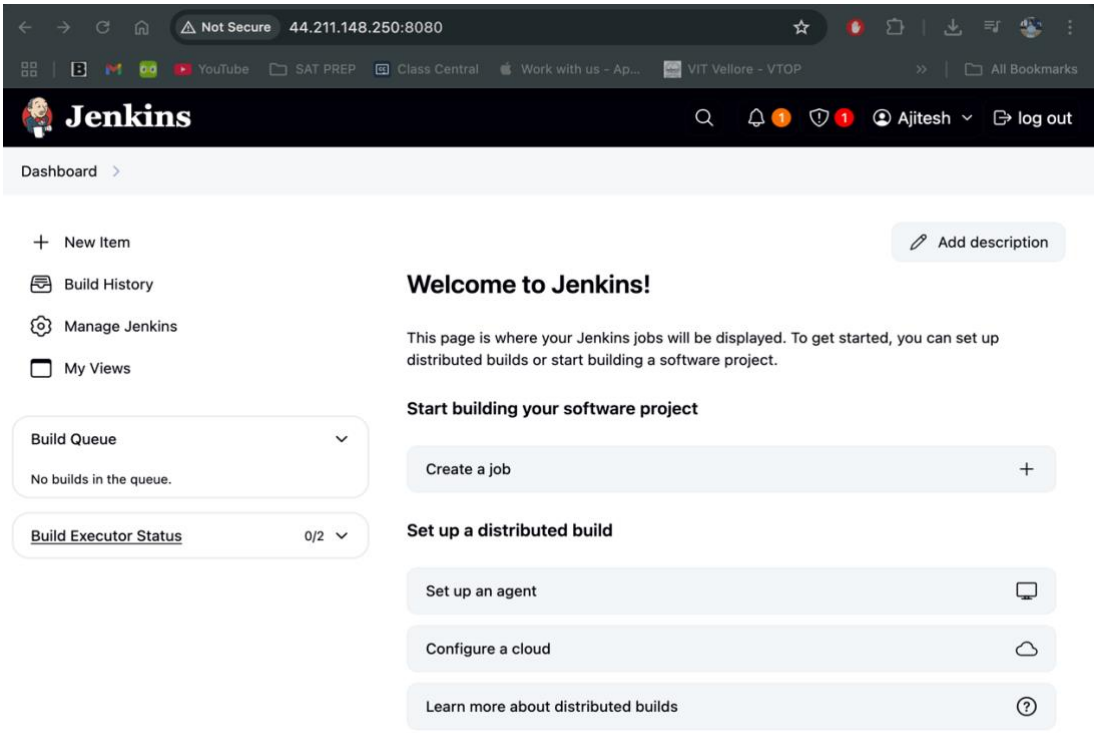
      3../<script_name>.sh to execute it.



2. **Install Jenkins (jenkins.sh)**
    - **Purpose:** Jenkins serves as the central automation engine that orchestrates the entire CI/CD pipeline. This script installs Jenkins and its required dependency, Java, which provides the runtime environment for Jenkins.

```
#!/bin/bash

# Install OpenJDK 17 JRE Headless
sudo apt install openjdk-17-jre-headless -y

# Download Jenkins GPG key
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

# Add Jenkins repository to package manager sources
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
  https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null

# Update package manager repositories
sudo apt-get update

# Install Jenkins
sudo apt-get install jenkins -y
~
~
```

```
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
ubuntu@ip-172-31-87-106:~$ systemctl status jenkins
● jenkins.service - Jenkins Continuous Integration Server
     Loaded: loaded (/usr/lib/systemd/system/jenkins.service; enabled; preset: enabled)
     Active: active (running) since Thu 2025-06-19 11:46:04 UTC; 35min ago
   Main PID: 3530 (java)
      Tasks: 45 (limit: 9501)
     Memory: 1015.4M (peak: 1.0G)
        CPU: 24.404s
     CGroup: /system.slice/jenkins.service
             └─3530 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --w
ebroot=/var
```



3. **Install Docker (docker.sh)**
    - **Purpose:** Docker is essential for packaging the application into a portable, lightweight container. This script installs the Docker engine.
    - sudo chmod 666 /var/run/docker.sock: This post-installation command grants necessary permissions to interact with the Docker daemon without requiring sudo for every command, streamlining operations.
    - docker login -u aj2005: This command was executed to log into the Docker Hub account, which is a prerequisite for pushing the application's Docker image to a public or private repository later in the pipeline.

```
ubuntu@ip-172-31-87-106:~$ cat docker.sh
#!/bin/bash

# Update package manager repositories
sudo apt-get update

# Install necessary dependencies
sudo apt-get install -y ca-certificates curl

# Create directory for Docker GPG key
sudo install -m 0755 -d /etc/apt/keyrings

# Download Docker's GPG key
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc

# Ensure proper permissions for the key
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add Docker repository to Apt sources
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.c
om/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Update package manager repositories
sudo apt-get update

sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

```
Processing triggers for man-db (2.12.0-4build2) ...
Processing triggers for libc-bin (2.39-0ubuntu8.4) ...
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
ubuntu@ip-172-31-87-106:~$ docker --version
Docker version 28.2.2, build e6534b4
ubuntu@ip-172-31-87-106:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
     Active: active (running) since Thu 2025-06-19 12:31:02 UTC; 19min ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 5002 (dockerd)
      Tasks: 9
     Memory: 20.8M (peak: 22.4M)
        CPU: 461ms
     CGroup: /system.slice/docker.service
             └─5002 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

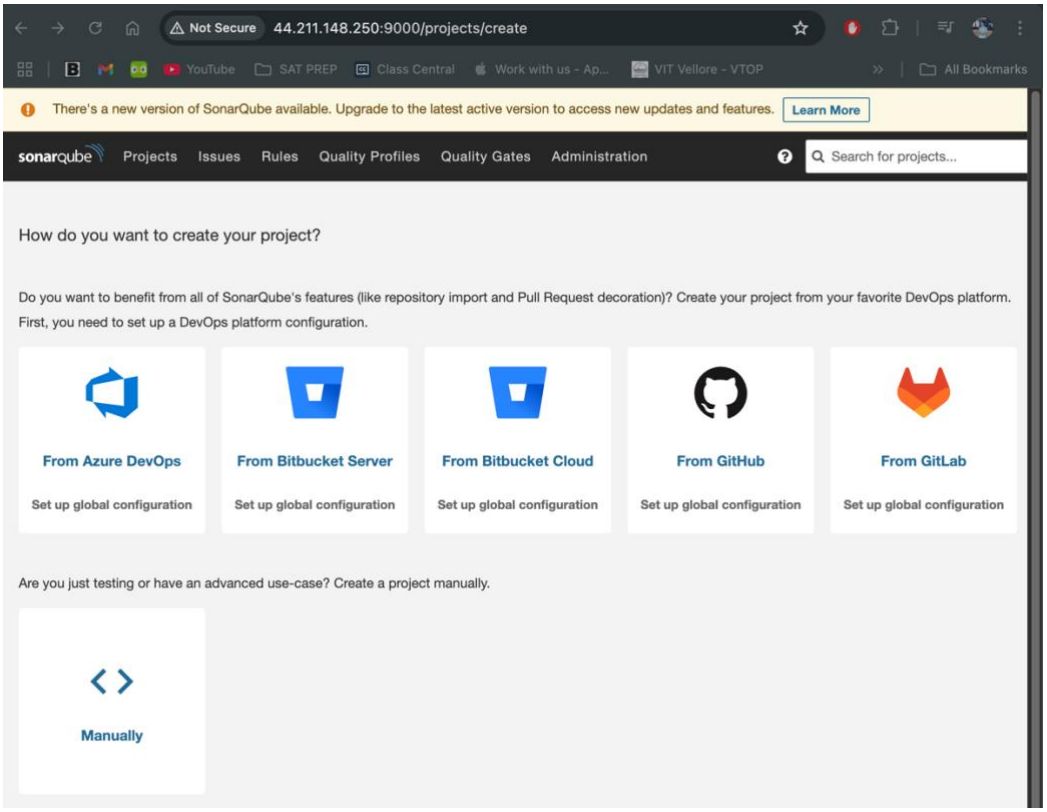
```
ubuntu@ip-172-31-87-106:~$ docker login -u aj2005

i Info → A Personal Access Token (PAT) can be used instead.
         To create a PAT, visit https://app.docker.com/settings


Password:

WARNING! Your credentials are stored unencrypted in '/home/ubuntu/.docker/config.json'.
Configure a credential helper to remove this warning. See
https://docs.docker.com/go/credential-store/

Login Succeeded
ubuntu@ip-172-31-87-106:~$
```

4. **Install Trivy (trivy.sh)**
   - **Purpose:** Trivy is a simple and comprehensive security scanner. It will be utilized to identify vulnerabilities in the project's files, enhancing the security posture of the application.
   - trivy --version: After installation, this command was run to verify that Trivy was correctly installed and ready for use.

```
ubuntu@ip-172-31-87-106:~$ cat trivy.sh
#!/bin/bash
sudo apt-get install wget apt-transport-https gnupg
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | gpg --dearmor | sudo tee /usr/share/keyr
ings/trivy.gpg > /dev/null
echo "deb [signed-by=/usr/share/keyrings/trivy.gpg] https://aquasecurity.github.io/trivy-repo/deb generic main
" | sudo tee -a /etc/apt/sources.list.d/trivy.list
sudo apt-get update
sudo apt-get install trivy
```

5. **Setup SonarQube**
   - **Purpose:** SonarQube performs static code analysis to detect bugs, code smells, and security vulnerabilities within the application code, improving code quality and maintainability. It was run as a Docker container for a quick and isolated setup.
   - docker run -d --name sonar -p 9000:9000 sonarqube:lts-community: This command downloads the sonarqube:lts-community image and starts it as a detached (-d) container named sonar. It maps port 9000 on the host to port 9000 within the container, making the SonarQube UI accessible via the server's IP address.

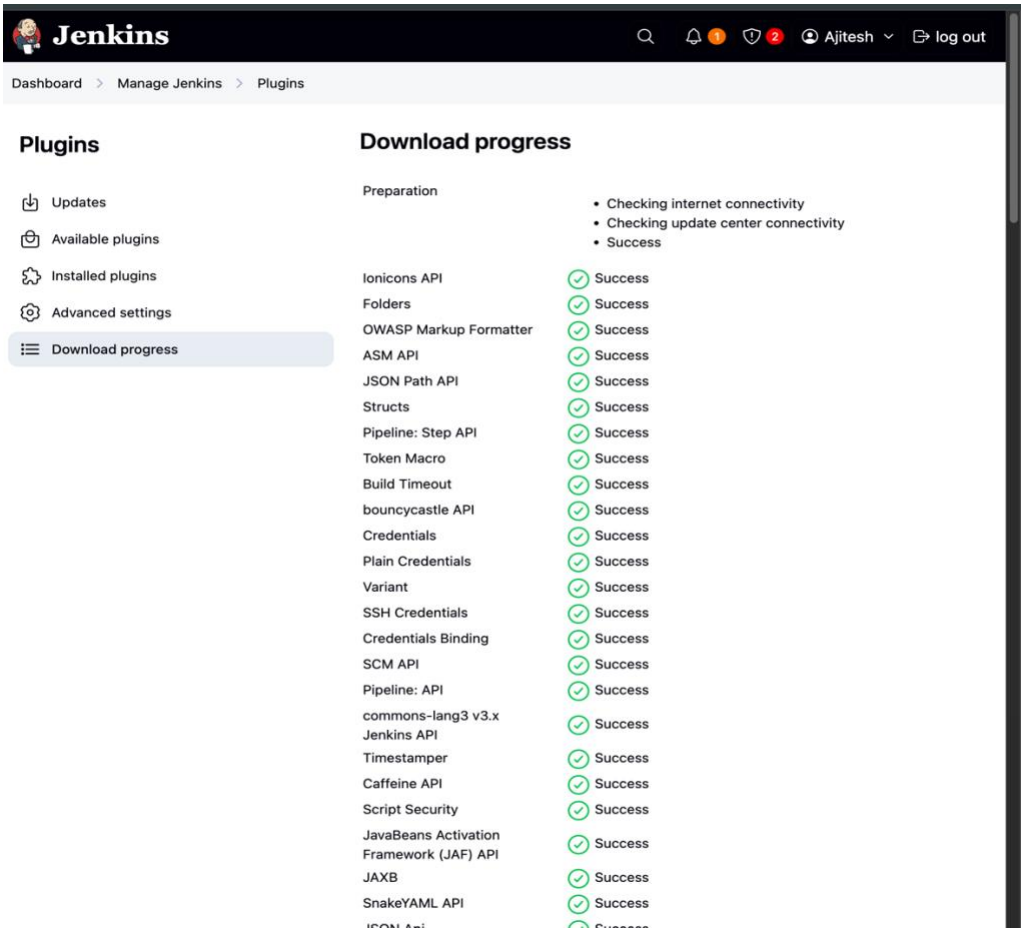
```
ubuntu@ip-172-31-87-106:~$ docker run -d --name sonar -p 9000:9000 sonarqube:lts-community
Unable to find image 'sonarqube:lts-community' locally
lts-community: Pulling from library/sonarqube
89dc6ea4eae2: Pull complete
31436012ac5b: Pull complete
2d16eb76e762: Pull complete
ac81863d97cb: Pull complete
26f6dfeccc10: Pull complete
a7343cdb8fb1: Pull complete
7c8e090ec954: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:1f0acc4139fc0bf09f39b962a63b56b10393cb0d07a4aa28346ea0af5b95f764
Status: Downloaded newer image for sonarqube:lts-community
3025a0e31d3e0a2448eabb5e12302bc9677693cfaa4e4cbeeb116f41a0322b14
ubuntu@ip-172-31-87-106:~$ docker images
REPOSITORY   TAG             IMAGE ID       CREATED       SIZE
sonarqube    lts-community   744b8bba9022   3 weeks ago   604MB
```



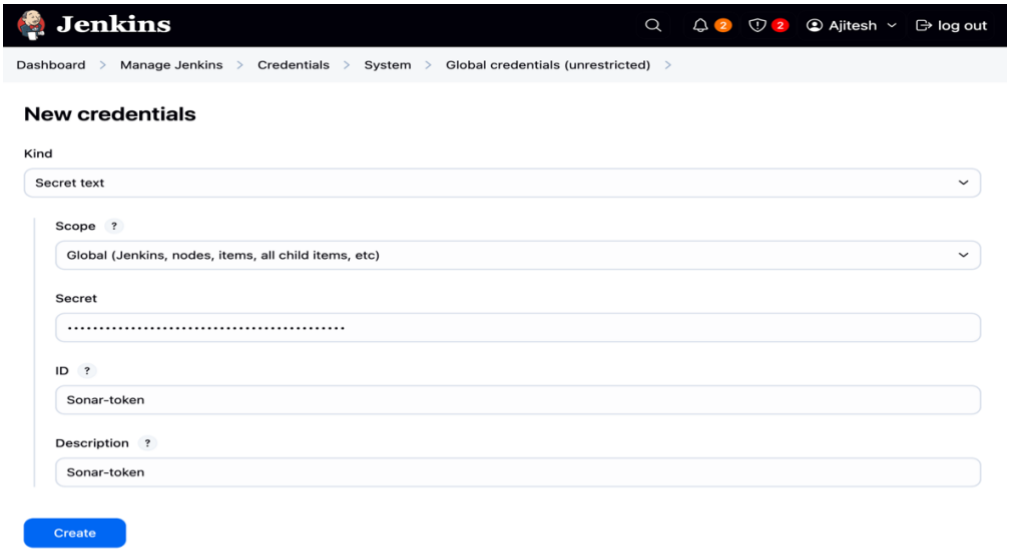
Step 3 & 4: Jenkins and Email Configuration

Jenkins was then configured to orchestrate the pipeline and send notifications regarding build statuses.

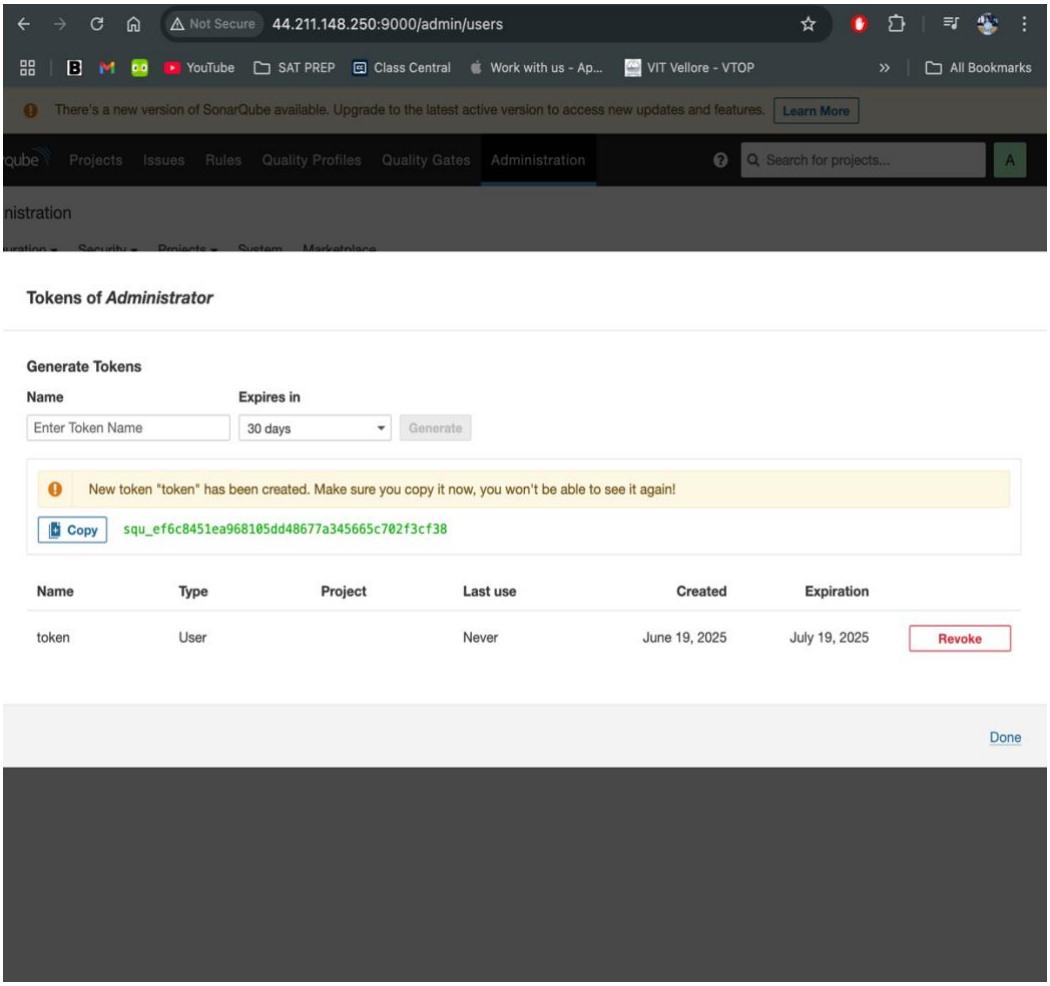
1. **Initial Jenkins Setup**
    - The Jenkins dashboard was navigated to at http://44.211.148.250:8080 *(Was deployed during build, currently inactive)*.
    - **Install Plugins:** From **Manage Jenkins > Plugins**, recommended plugins were installed, including SonarQube scanner, Docker Pipeline, Kubernetes, and Email Extension Template. These plugins extend Jenkins' core capabilities, enabling it to interact seamlessly with other DevOps tools and platforms.

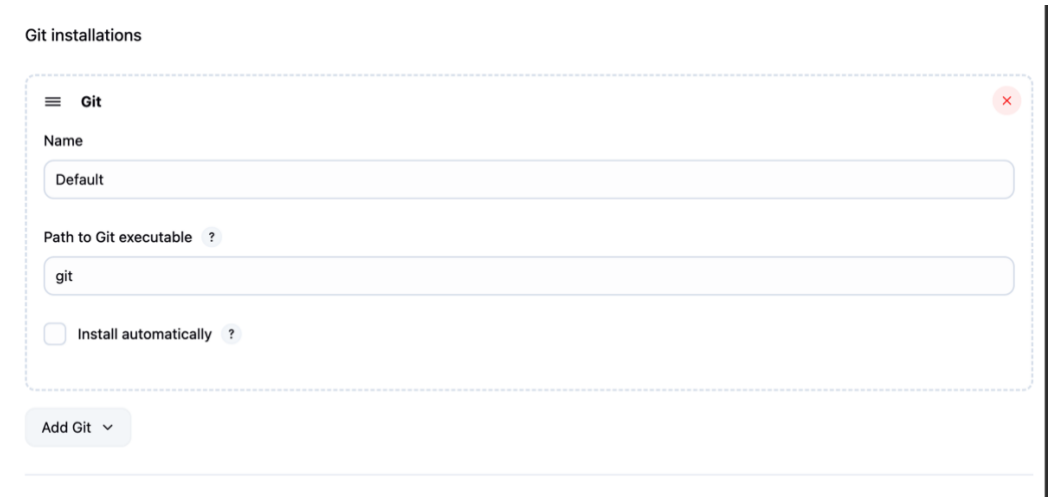


- **Configure Tools and Credentials:** Paths for JDK and SonarQube Scanner were configured by navigating to **Manage Jenkins > Tools**. Additionally, Docker Hub password and SonarQube token were securely stored as credentials under **Manage Jenkins > Credentials**, ensuring sensitive information is not exposed directly in scripts.



Generating token via Sonarqube:

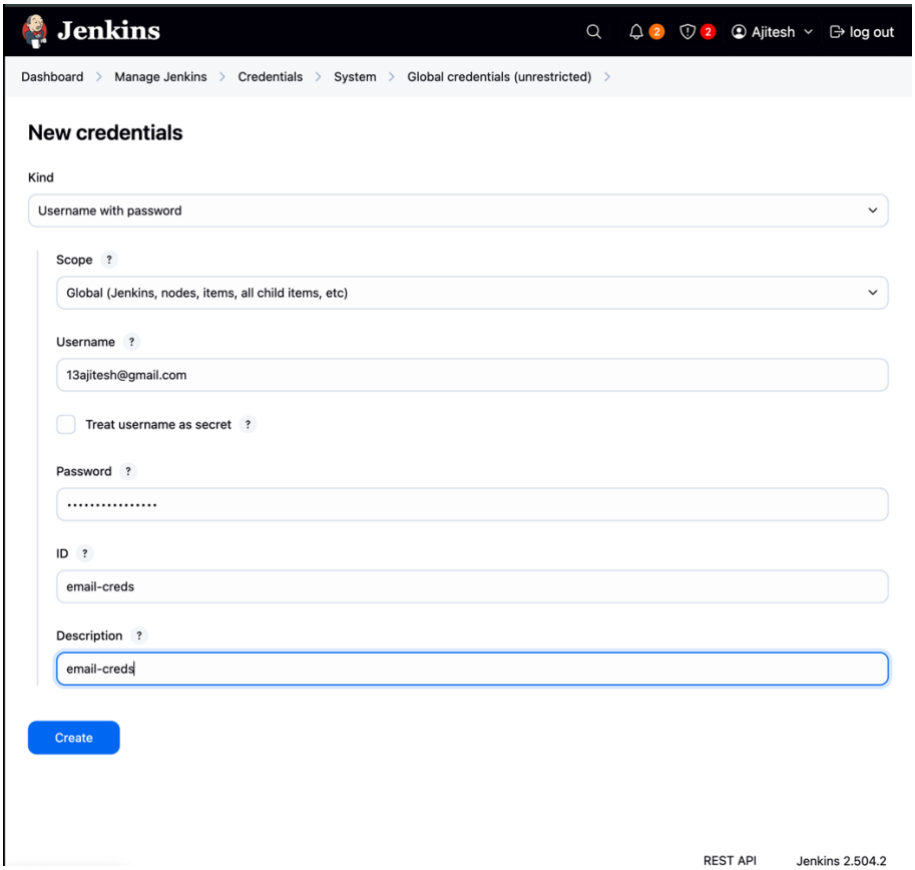




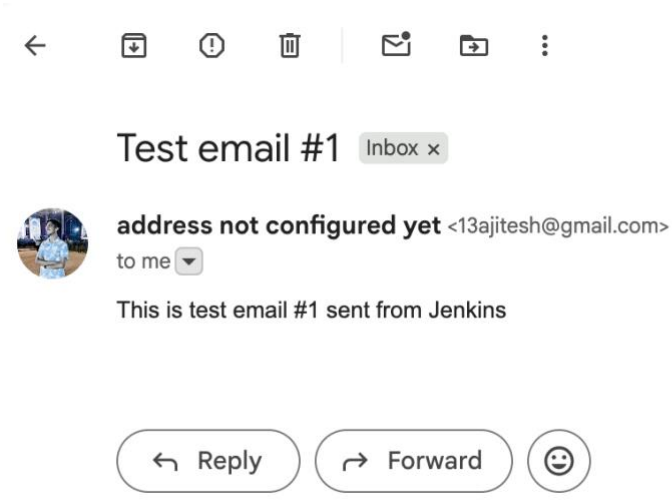
2. **Configure Email Notifications**
    - An "App Password" was generated from the Google Account settings. This is necessary for external applications like Jenkins to send emails via a Google account.

- In **Manage Jenkins > System**, the **Extended E-mail Notification** and SMTP server (smtp.gmail.com), port (465), and adding the generated App Password credentials.

  This setup enables Jenkins to automatically send status emails after each build completes, keeping management and developers informed.



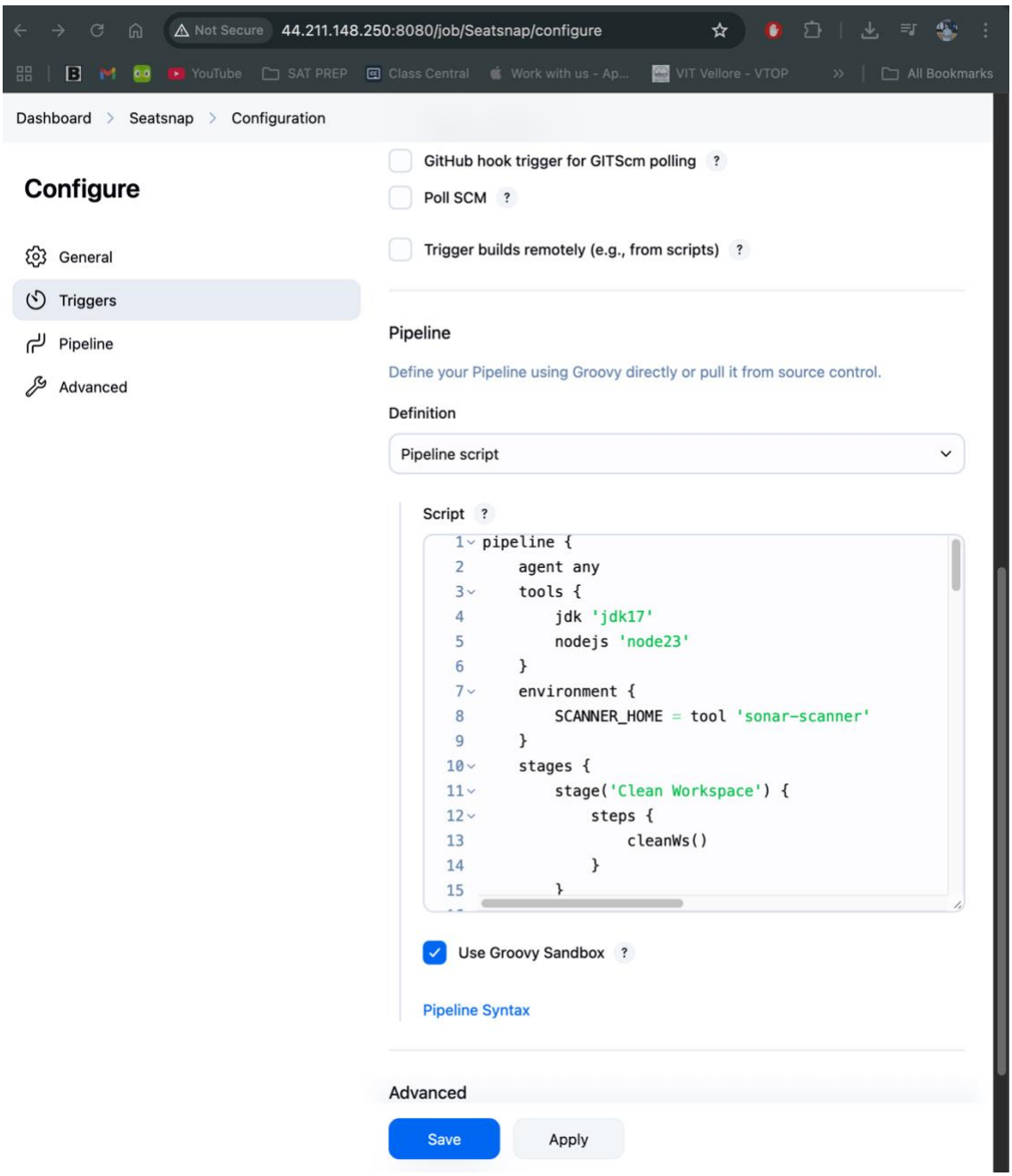
Sample Mail Prototype using Jenkins:



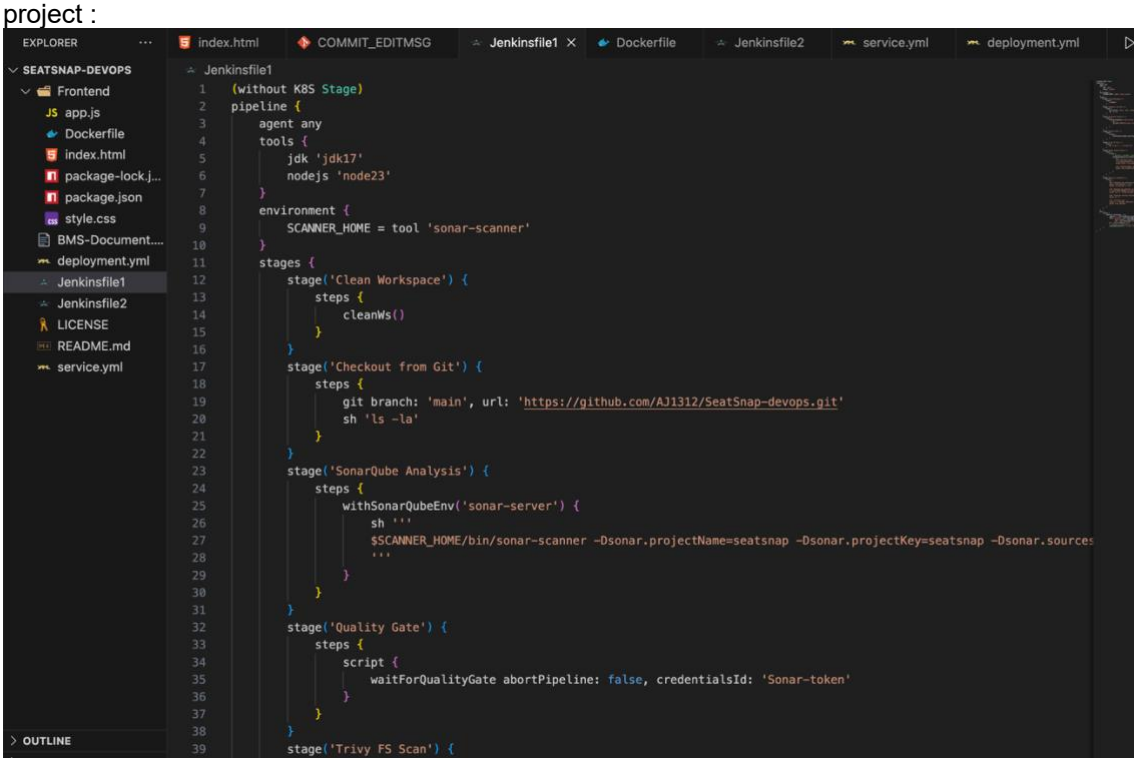
Step 5: Create the Jenkins Pipeline

This step defines the automated workflow that governs the build, test, and deployment process.

1. **Create Pipeline Job**
    - In Jenkins, a new "**Pipeline**" project was created.
    - The following Groovy script was pasted into the "**Pipeline script**" section. This script defines a series of stages, outlining the comprehensive steps from code checkout to container deployment.



Ensure Jenkinfiles on your github repo, we have **jenkinsfile1** and **jenkinsfile 2**, file1 contains pipeline without Kubernetes while 2 contains Kubernetes deployment as shown later in the project :



- **Run the Pipeline:** The pipeline was saved, and "**Build Now**" was clicked to initiate the automated process. Progress could be monitored in the "**Stage View**" to track each stage's execution. Upon successful completion, the Docker image was confirmed to be successfully built and pushed to the Docker Hub repository (aj2005/snapseat:latest). The SnapSeat application could then be accessed at http://<SnapSeat-Server-IP>:3000, confirming successful container deployment.

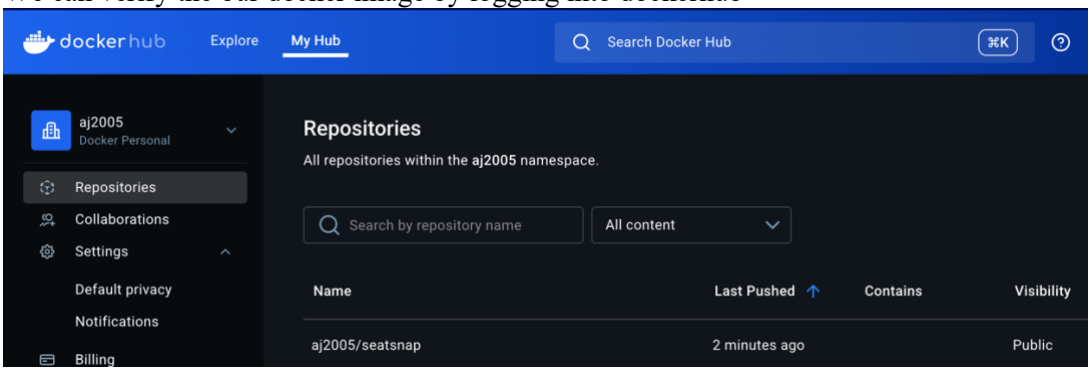
We were able to run our pipeline successfully in Jenkins:

Seatsnap

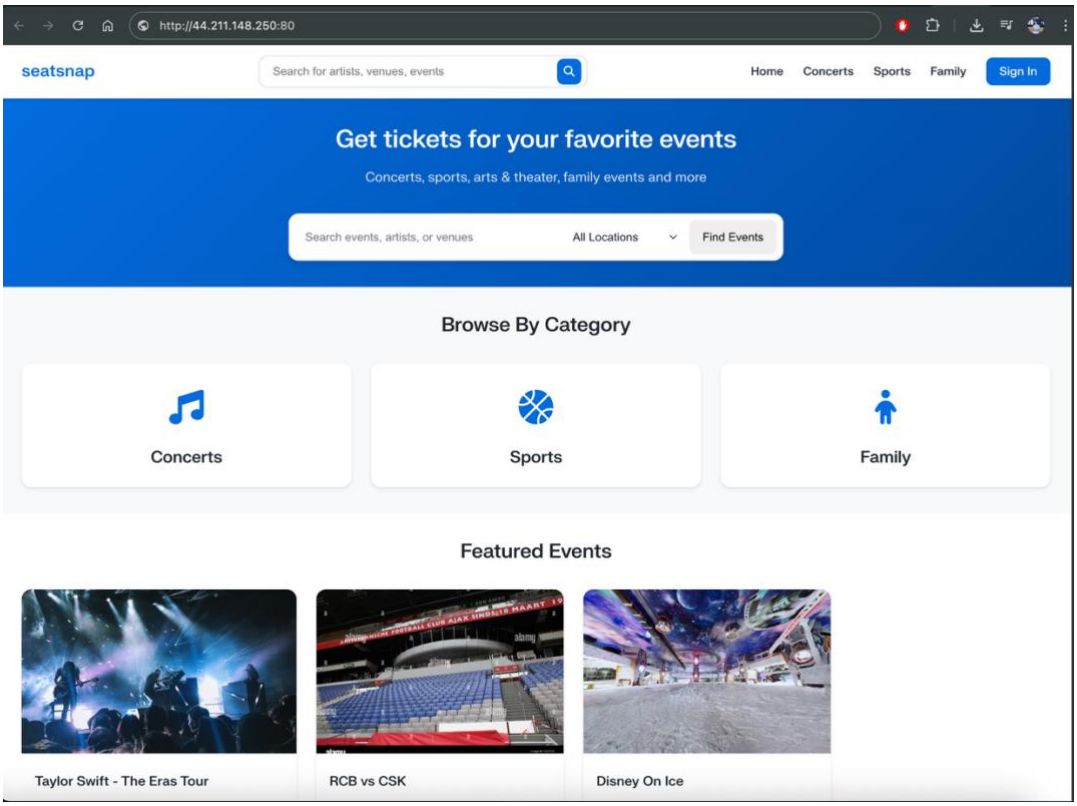
Add description

Declarative: Tool Install	Clean Workspace	Checkout from Git	SonarQube Analysis	Quality Gate	Trivy FS Scan	Docker Build & Push	Deploy to Container	Declarative: Post Actions
171ms	299ms	880ms	9s	287ms	421ms	2min 23s	69ms	325ms
156ms	274ms	855ms	18s	502ms (paused for 1s)	771ms	2min 23s	7s	313ms

- We can verify the our docker image by logging into dockerhub



- Successful build indicates that our pipeline is successfully deployed and our site is up and running
  Was deployed at URL generated using AWS. (*Inactive currently)*



Summary of Part I: Docker Deployment Part I established the foundational environment for the SnapSeat application's Docker deployment. This involved setting up an AWS EC2 instance, configuring IAM roles for secure access, and installing essential DevOps tools like Jenkins, Docker, Trivy, and SonarQube. The Jenkins pipeline was meticulously crafted to automate the entire CI/CD process, from code checkout and quality analysis to Docker image building, pushing, and finally deploying the application as a Docker container. This phase ensures the application is containerized, scanned for vulnerabilities, and automatically deployed to a single server environment.

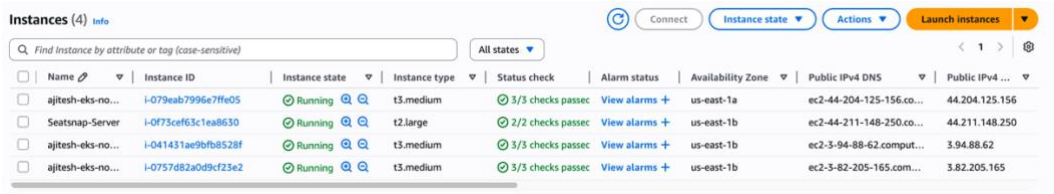
Part II: Kubernetes Deployment & Monitoring

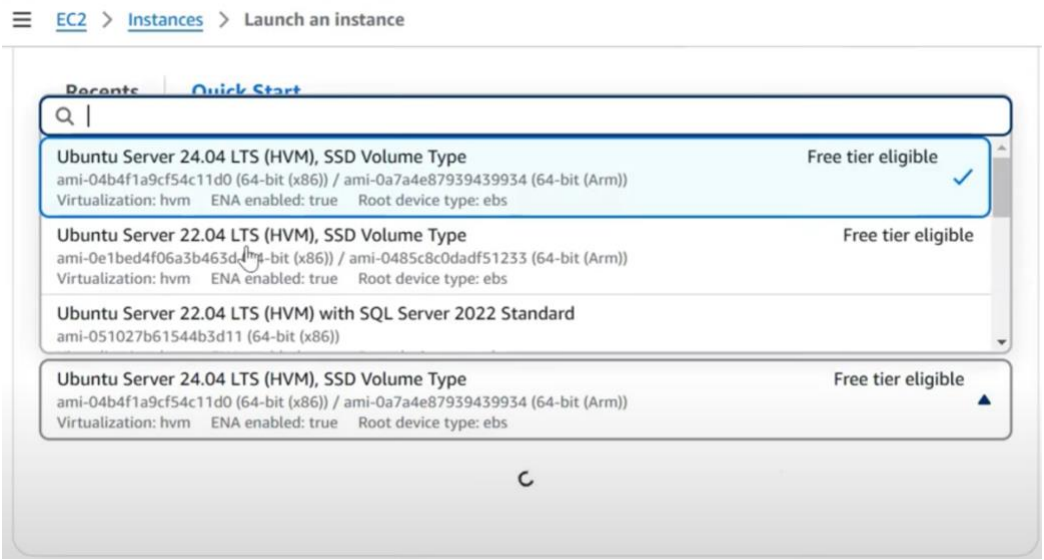
In this part, the application will be deployed to a scalable Kubernetes cluster, and comprehensive monitoring will be set up to observe its health and performance in a distributed environment.

Step 1: Create an EKS Cluster

The SnapSeat-Server will be leveraged to create a Kubernetes cluster in AWS, providing a robust and scalable infrastructure for the application.

1. **Install EKS Command-Line Tools**
    - **aws-cli**, **kubectl**, and **eksctl** were installed on the SnapSeat-Server. These tools are indispensable for interacting with AWS services, managing Kubernetes clusters, and provisioning EKS clusters, respectively.
    - `aws configure`: This command was run, and the Access Keys for the `ajitesh-eks-user` were provided. This configures the AWS CLI to use the credentials of the IAM user specifically created for EKS operations.





2. **Provision the EKS Cluster**
    - `eksctl` was used to create the cluster components in a specific, sequential order:
    - `eksctl create cluster --name=ajitesh-eks ... --without-nodegroup`: This initial command created the EKS control plane, often referred to as the "brain" of the Kubernetes cluster. It was provisioned without any worker nodes at this stage to allow for separate configuration.

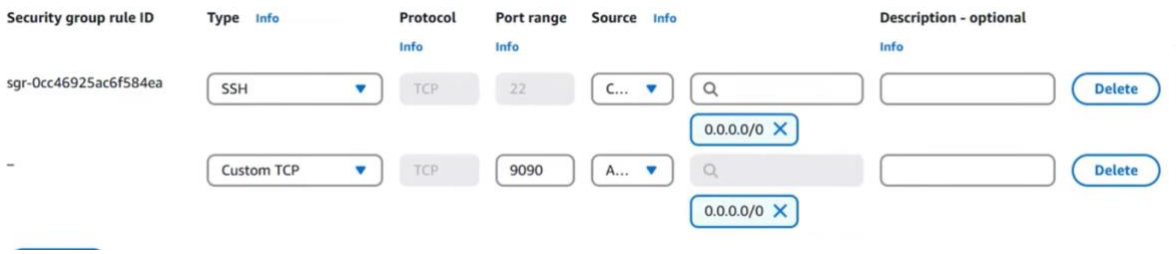
- eksctl utils associate-iam-oidc-provider ...: This crucial step enabled IAM Roles for Service Accounts (IRSA). IRSA is a security feature that allows Kubernetes pods to securely access other AWS services using IAM roles, rather than requiring AWS credentials to be stored directly within the pods.
- eksctl create nodegroup --cluster=ajitesh-eks ...: This command created the worker nodes. These are the EC2 instances where the application containers will actually run, forming the compute capacity of the Kubernetes cluster.

Step 2: Update Jenkins Pipeline for EKS

Next, a final deployment stage was incorporated into the existing Jenkins pipeline to facilitate deployment to the Kubernetes cluster.

1. **Configure Jenkins for EKS Access**
   - The jenkins user on the server requires appropriate permissions to communicate with AWS services, specifically the EKS cluster.
   - sudo -su jenkins: Temporarily switched to the jenkins user. This step ensures that subsequent commands are executed with the permissions of the Jenkins service account.
   - aws configure: AWS credentials were configured for this user. This allows the Jenkins user to authenticate with AWS for EKS operations.
   - aws eks update-kubeconfig --name ajitesh-eks ...: This command was run to configure kubectl for the jenkins user. It updates the kubeconfig file, enabling kubectl to connect and manage the newly created EKS cluster.
2. **Add the 'Deploy to EKS' Stage**
   - The following stage was added to the Jenkins pipeline script, just before the post section. This stage leverages kubectl commands to apply the application's deployment and service configurations to the EKS cluster.

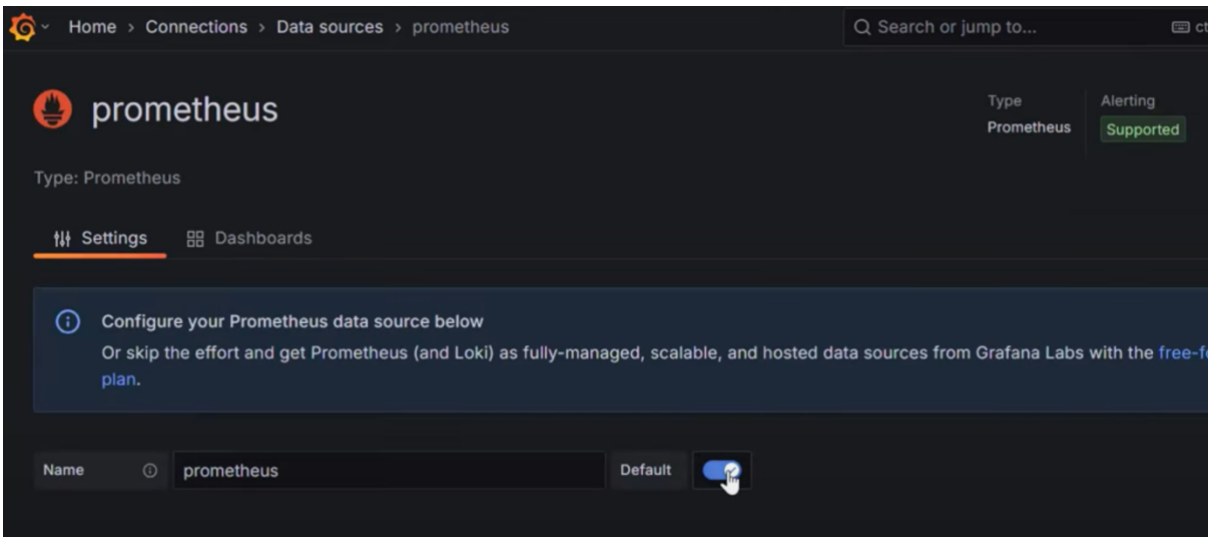
- The updated pipeline was run. After a successful execution, kubectl get svc was used to retrieve the Load Balancer URL. This external URL provides access to the application now running on Kubernetes, demonstrating the successful deployment to the cluster.



Step 3: Monitoring with Prometheus and Grafana

Finally, a robust monitoring solution was established using Prometheus for data collection and Grafana for visualization.

1. **Launch a new "Monitoring-Server" EC2 instance.** This separate instance is dedicated to hosting the monitoring tools, isolating them from the application servers and ensuring monitoring capabilities even if application instances face issues.
2. **Install Prometheus**: Prometheus, a powerful open-source monitoring system and time-series database, was installed. Script steps were followed to create a prometheus user, download and configure Prometheus, and then run it as a service to continuously collect metrics.




```
ubuntu@ip-172-31-18-218:~$ tar -xvf prometheus-2.47.1.linux-amd64.tar.gz
prometheus-2.47.1.linux-amd64/
prometheus-2.47.1.linux-amd64/LICENSE
prometheus-2.47.1.linux-amd64/NOTICE
prometheus-2.47.1.linux-amd64/prometheus.yml
prometheus-2.47.1.linux-amd64/consoles/
prometheus-2.47.1.linux-amd64/consoles/prometheus.html
prometheus-2.47.1.linux-amd64/consoles/prometheus-overview.html
prometheus-2.47.1.linux-amd64/consoles/node-cpu.html
prometheus-2.47.1.linux-amd64/consoles/index.html.example
prometheus-2.47.1.linux-amd64/consoles/node.html
prometheus-2.47.1.linux-amd64/consoles/node-disk.html
prometheus-2.47.1.linux-amd64/consoles/node-overview.html
prometheus-2.47.1.linux-amd64/promtool
prometheus-2.47.1.linux-amd64/console_libraries/
prometheus-2.47.1.linux-amd64/console_libraries/prom.lib
prometheus-2.47.1.linux-amd64/console_libraries/menu.lib
prometheus-2.47.1.linux-amd64/prometheus
```

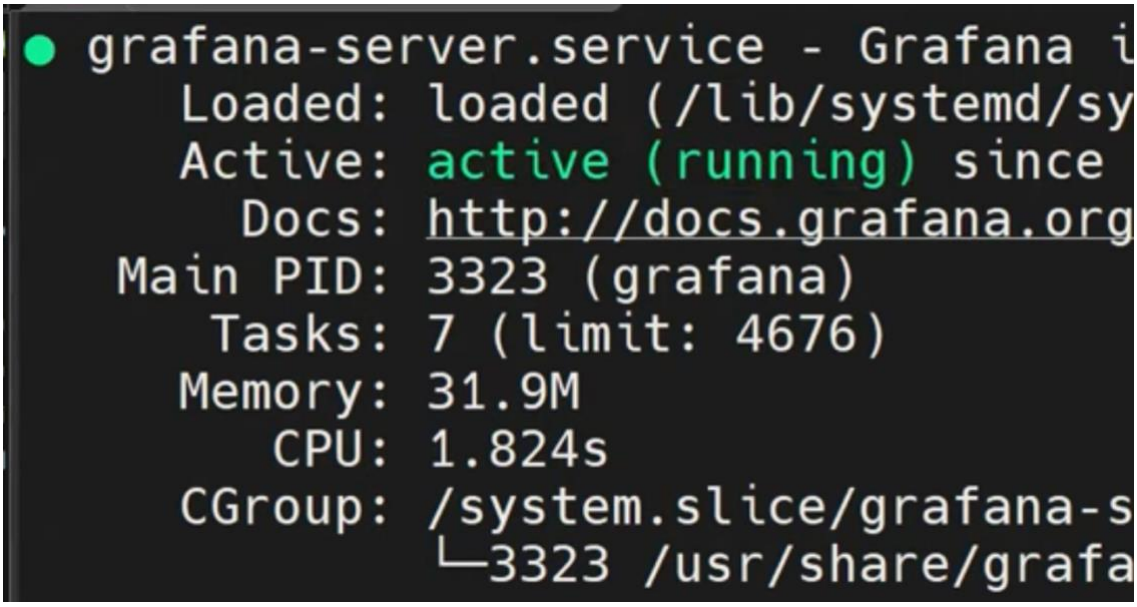
3. **Install Node Exporter**: Node Exporter, an agent designed to collect system-level metrics (such as CPU usage, RAM consumption, and disk I/O), was installed on both the **Monitoring-Server** and **SnapSeat-Server**. This enables Prometheus to scrape performance data from both instances.

```
[Unit]
Description=Node Exporter
Wants-network-online.target
After network-online.target
StartLimitIntervalSec=500
StartLimitBurst=5
[Service]
User=node_exporter
Group=node_exporter
Type=simple
Restart=on-failure
RestartSec=5s
ExecStart=/usr/local/bin/node_exporter --collector.logind
[Install]
WantedBy=multi-user.target
```

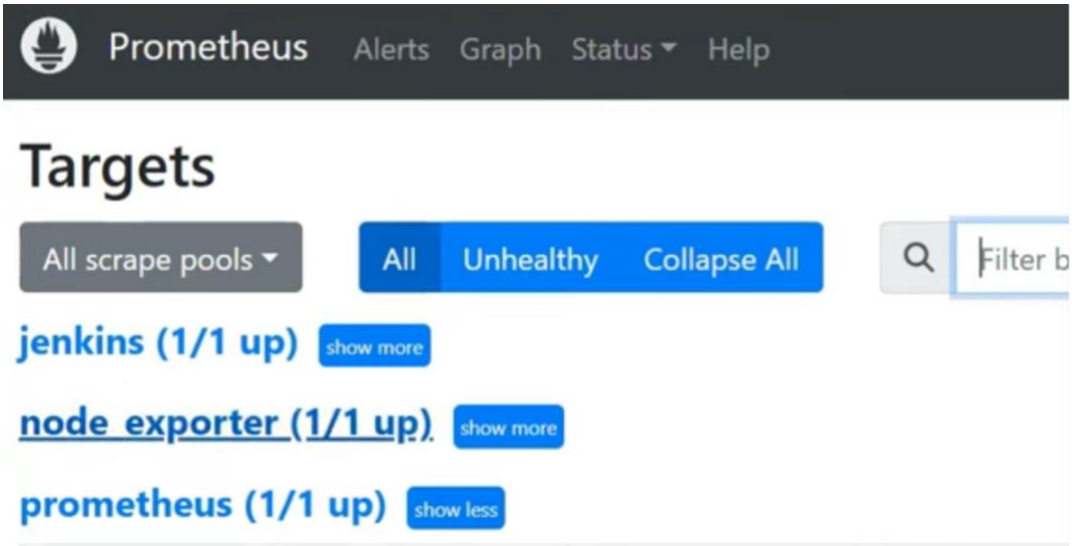
4. **Configure Prometheus Targets**: The /etc/prometheus/prometheus.yml configuration file was edited. This file instructs Prometheus where to scrape metrics from, including itself, the Node Exporters running on both servers, and the Jenkins /prometheus endpoint, ensuring comprehensive data collection.

```
After=network online.target
StartLimitIntervalSec=500
StartLimitBurst=5
[Service]
User=prometheus
Group=prometheus
Type=simple
Restart=on-failure
RestartSec=5s
ExecStart=/usr/local/bin/prometheus \
--config.file=/etc/prometheus/prometheus.yml \
-storage.tsdb.path=/data \
--web.console.templates=/etc/prometheus/consoles \
--web.console.libraries=/etc/prometheus/console_libraries \
--web.listen-address=0.0.0.0:9090 \
--web.enable-lifecycle
[Install]
WantedBy=multi-user.target
```

5. **Install Grafana**: Grafana, an open-source platform for analytics and interactive visualization, was installed on the **Monitoring-Server**. It provides a user-friendly interface to query, visualize, alert on, and understand metrics.



6. **Configure Grafana**: Grafana was accessed.
    - Prometheus (http://localhost:9090) was added as a data source. This connects Grafana to the Prometheus server, allowing it to retrieve collected metrics.
    - Dashboards were imported by navigating to **Create > Import** and using dashboard IDs from Grafana's official site. This provides instant, pre-built visualizations for common metrics. Good starting points, like **1860 (Node Exporter Full)** and **9964 (Jenkins Performance)**, offer immediate insights into server health and Jenkins pipeline performance.



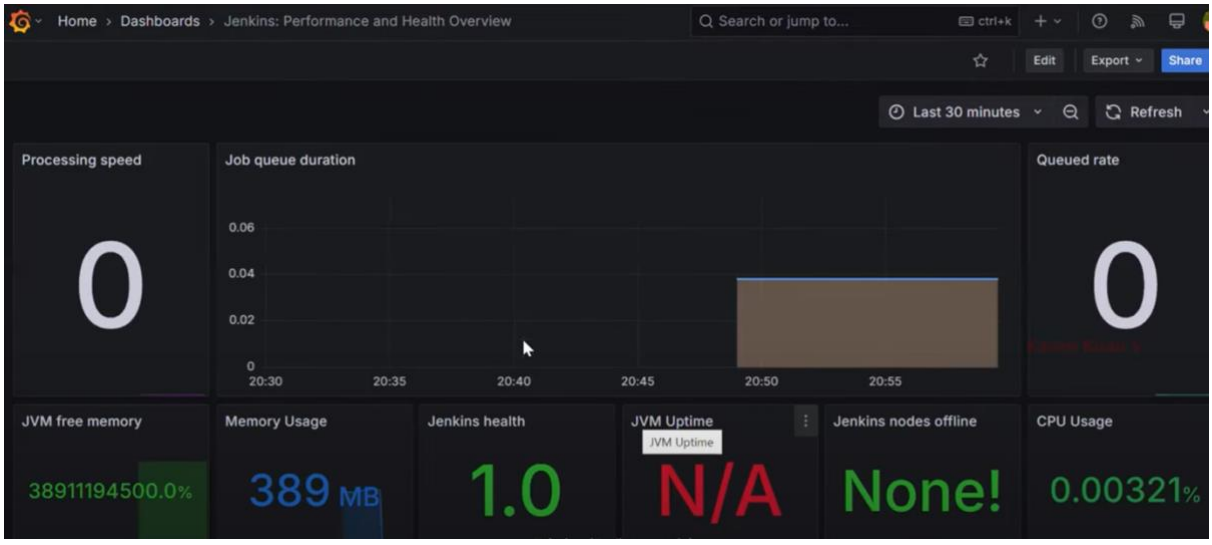
Result: Centralized Monitoring in Grafana

By completing this setup, you have successfully established a powerful, real-time monitoring solution. Prometheus is now actively scraping performance metrics from the Jenkins server (monitoring your CI/CD pipeline's health) and system metrics from the Node Exporter (tracking CPU, memory, and disk usage of your servers).

In the Grafana web interface, the imported dashboards are now populated with live data. You can see dynamic graphs and charts visualizing the health of your infrastructure and the performance of your Jenkins builds, providing immediate insight and enabling proactive system management.

NODE EXPORTER:

JENKINS:



Summary of Part II: Kubernetes Deployment & Monitoring Part II transitioned the SnapSeat application from a single Docker container deployment to a scalable Kubernetes environment on AWS EKS, complete with robust monitoring. This involved setting up the EKS cluster using eksctl, updating the Jenkins pipeline to automate deployment to Kubernetes using kubectl, and configuring Jenkins for secure EKS access. Finally, a comprehensive monitoring stack was established using Prometheus for metric collection and Grafana for insightful visualizations, running on a dedicated monitoring server. This phase ensures the application is highly available, scalable, and continuously monitored for performance and health.

Final Reflection

This project, *SeatSnap – A Ticketmaster-Style Site with DevOps Implementation*, provided an invaluable opportunity to apply modern DevOps principles in a real-world scenario. From setting up a CI/CD pipeline using Jenkins, Docker, and Kubernetes, to implementing monitoring solutions with Prometheus and Grafana, the project showcased the power of automation in software delivery.

Through this work, I gained hands-on experience in:

- **Linux fundamentals** for server setup, automation, and tool installation
- **AWS infrastructure management** including EC2, EKS, IAM, and networking configurations
- **Containerization and orchestration** using Docker and Kubernetes
- **Continuous integration, delivery, and monitoring** of applications

These experiences have strengthened my understanding of scalable, resilient architectures and the crucial role DevOps plays in modern software engineering. The project also taught me the importance of automation, security, and continuous improvement in the development lifecycle.

Conclusion

The *SeatSnap* project successfully demonstrated the implementation of a fully automated DevOps pipeline for deploying and managing a web application. The project achieved:

- Automated build, testing, security scanning, and deployment using Jenkins, SonarQube, Trivy, and Docker.
- Deployment to a Kubernetes cluster (AWS EKS) for scalability and resilience.
- Comprehensive monitoring through Prometheus and Grafana for real-time observability.

The current solution focuses on the frontend and infrastructure automation, and it lays the groundwork for future enhancements — including backend integration, persistent storage, Infrastructure-as-Code, and advanced security measures.

I would like to sincerely thank **IBM (Adroit Technologies)** for providing the *IBM DevOps* course, which guided this project's direction. I am also grateful to **Vellore Institute of Technology, Vellore** for offering this platform and encouraging practical, hands-on learning in emerging technologies.